In [6]:

```
import pandas as pd
import numpy as np
# Load dataset from a CSV file
df = pd.read_csv('data 2.csv')
df2=pd.read_csv('data 5.csv')
```

In [7]:

```
df
```

Out[7]:

	User_Id	Place_Id	Place_rating
0	5	1	4.1
1	40	2	4.2
2	11799	3	4.6
3	81	4	3.1
4	69	5	3.7
...	...	...	...
165	6	166	4.3
166	9	167	3.8
167	48	168	3.9
168	18154	169	4.1
169	4	170	3.5

170 rows × 3 columns

In [8]:

```
df2
```

Out[8]:

	Place_Id	Source	Destination	Distance(km)
0	1	Amtala	Bishnupur	2.20
1	2	Bishnupur	Khoriberia	4.60
2	3	Khoriberia	Vasa Mandir	1.50
3	4	Vasa mandir	Pailan	5.40
4	5	Pailan	Joka	5.00
...	...	...	...	...
165	166	BK Pal(Rabindra Sarani)	Ahiritola	1.20
166	167	Ahiritola	Jora Bagan	0.85
167	168	Jora Bagan	Mala para	0.21
168	169	Mala para	Satyanarayan Park	1.30
169	170	Satyanarayan Park	Barabazar	0.50

170 rows × 4 columns

In [9]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 170 entries, 0 to 169
Data columns (total 3 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   User_Id       170 non-null    int64  
 1   Place_Id      170 non-null    int64  
 2   Place_rating  170 non-null    float64
dtypes: float64(1), int64(2)
memory usage: 4.1 KB
```

In [10]:

```
df2.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 170 entries, 0 to 169
Data columns (total 4 columns):
 #   Column        Non-Null Count  Dtype  
---  ------        --------------  -----  
 0   Place_Id      170 non-null    int64  
 1   Source        170 non-null    object 
 2   Destination   170 non-null    object 
 3   Distance(km)  170 non-null    float64
dtypes: float64(1), int64(1), object(2)
memory usage: 5.4+ KB
```

In [ ]:

```
```

In [ ]:

```
```

In [13]:

```
import csv
import heapq
```

```python
import ipywidgets as widgets
from IPython.display import display, clear_output

# Class representing the graph
class Graph:
    def __init__(self, graph_dict=None, directed=True):
        self.graph_dict = graph_dict or {}
        self.directed = directed
        if not directed:
            self.make_undirected()

    # Method to make the graph undirected by adding reverse edges
    def make_undirected(self):
        for a in list(self.graph_dict.keys()):
            for (b, dist) in self.graph_dict[a].items():
                self.graph_dict.setdefault(b, {})[a] = dist

    # Method to connect two places with a given distance
    def connect(self, A, B, distance=1):
        self.graph_dict.setdefault(A, {})[B] = distance
        if not self.directed:
            self.graph_dict.setdefault(B, {})[A] = distance

    # Method to get neighbors of a place
    def get(self, a, b=None):
        links = self.graph_dict.setdefault(a, {})
        if b is None:
            return links
        else:
            return links.get(b)

# Function to read data from CSV file
def read_csv(filename):
    with open(filename, 'r', encoding='utf-8') as csvfile:
        reader = csv.DictReader(csvfile)
        data = [row for row in reader]
    return data

# Function to extract graph from the read CSV file's data
def extract_graph(data):
    graph = Graph(directed=False)  # Assuming the graph is undirected
    for row in data:
        origin = row['Place_Id']
        for compare_row in data:
            if origin != compare_row['Place_Id']:  # Connect all places
                graph.connect(origin, compare_row['Place_Id'], 1)  # Using a
    return graph

# A* search algorithm implementation
def astar_search(graph, start, end):
    open_list = []  # Priority queue to keep track of nodes to be explored
    heapq.heappush(open_list, (0, start, []))  # Initialize the priority que
    visited = set()  # Set to keep track of visited nodes

    while open_list:
        cost, current_place, path = heapq.heappop(open_list)  # Get the node
        if current_place == end:  # If the current node is the destination
            path.append(current_place)
            return path, cost

        if current_place not in visited:
            visited.add(current_place)  # Mark the current node as visited
            for neighbor, distance in graph.get(current_place).items():  # E
                total_cost = cost + distance  # Calculate total cost for the
                heapq.heappush(open_list, (total_cost, neighbor, path + [cur

    return None, float('inf')  # If no path is found

# Main function to create widgets and handle button clicks
def main():
    # Read data from CSV file and extract the graph
    data = read_csv('data 5.csv')
    graph = extract_graph(data)

    # Extract unique places from the dataset (using place_id)
    unique_places = sorted(set(row['Place_Id'] for row in data))
    places_list = [None] + unique_places  # Include None as the default opti

    # Create labels, dropdowns, buttons, and output widget using ipywidgets
    start_label = widgets.Label('Select Start Place ID:')
    start_place_dropdown = widgets.Dropdown(options=places_list)
    end_label = widgets.Label('Select Destination Place ID:')
    end_place_dropdown = widgets.Dropdown(options=places_list)

    calculate_button = widgets.Button(description='Calculate')
    output = widgets.Output()

    # Function to handle button click event
    def on_calculate_button_click(b):
        start_place = start_place_dropdown.value
        end_place = end_place_dropdown.value

        if start_place is None or end_place is None:
            with output:
                clear_output()
                print("Error: Please select both start and destination place
        elif start_place == end_place:
            with output:
                clear_output()
                print("Error: Start and destination places cannot be the sam
        else:
            # Call A* search algorithm and display the result
            path, total_distance = astar_search(graph, start_place, end_plac
            with output:
                clear_output()
                if path:
                    print("Shortest route from {} to {} is: {}".format(start
                    print("Total Path Cost (in units):", total_distance)
                else:
                    print("No path found.")
    # Attach the event handler function to the button click event
    calculate_button.on_click(on_calculate_button_click)
    # Display widgets using IPython display function
    display(start_label, start_place_dropdown, end_label, end_place_dropdowr

# Entry point of the program
if __name__ == "__main__":
    main()
```

```
Label(value='Select Start Place ID:')
Dropdown(options=(None, '1', '10', '100', '101', '102', '103', '104', '105',
'106', '107', '108', '109', '11',…
Label(value='Select Destination Place ID:')
Dropdown(options=(None, '1', '10', '100', '101', '102', '103', '104', '105',
'106', '107', '108', '109', '11',…
Button(description='Calculate', style=ButtonStyle())
Output()
```

1st method:

In [17]:

```
import csv
import heapq
import ipywidgets as widgets
from IPython.display import display, clear_output

# Class representing the graph
class Graph:
    def __init__(self, directed=True):
        self.graph_dict = {}
        self.directed = directed

    def connect(self, A, B, distance=1):
        self.graph_dict.setdefault(A, {})[B] = distance
        if not self.directed:
            self.graph_dict.setdefault(B, {})[A] = distance

    def get(self, a):
        return self.graph_dict.get(a, {})

# Function to read data from CSV file
def read_csv(filename):
    with open(filename, 'r', encoding='utf-8') as csvfile:
        return list(csv.DictReader(csvfile))

# Function to extract graph from the read CSV data
def extract_graph(data):
    graph = Graph(directed=False)
    for row in data:
        origin = row['Source']
        destination = row['Destination']
        if destination:
            distance = float(row['Distance(km)']) if row['Distance(km)'] els
            graph.connect(origin, destination, distance)
    return graph

# A* search algorithm implementation
```

```python
def astar_search(graph, start, end):
    open_list = []
    heapq.heappush(open_list, (0, start, []))
    visited = set()

    while open_list:
        cost, current_place, path = heapq.heappop(open_list)

        if current_place == end:
            return path + [current_place], cost

        if current_place not in visited:
            visited.add(current_place)
            for neighbor, distance in graph.get(current_place).items():
                total_cost = cost + distance
                heapq.heappush(open_list, (total_cost, neighbor, path + [cur

    return None, float('inf')

# Main function to create widgets and handle button clicks
def main():
    data = read_csv('data 5.csv')
    graph = extract_graph(data)

    unique_places = sorted(set(row['Source'] for row in data) | set(row['Des
    places_list = [None] + unique_places

    start_place_dropdown = widgets.Dropdown(options=places_list, descriptior
    end_place_dropdown = widgets.Dropdown(options=places_list, description='
    calculate_button = widgets.Button(description='Calculate')
    output = widgets.Output()

    def on_calculate_button_click(b):
        start_place = start_place_dropdown.value
        end_place = end_place_dropdown.value

        with output:
            clear_output()
            if start_place is None or end_place is None:
                print("Error: Please select both start and destination place
            elif start_place == end_place:
                print("Error: Start and destination places cannot be the sam
            else:
                path, total_distance = astar_search(graph, start_place, end_
                if path:
                    print(f"Shortest route from {start_place} to {end_place}
                    print(f"Total Path Cost (in Km): {total_distance}")
                else:
                    print("No path found.")

    calculate_button.on_click(on_calculate_button_click)
    display(start_place_dropdown, end_place_dropdown, calculate_button, outp

# Entry point of the program
if __name__ == "__main__":
    main()
```

```
Dropdown(description='Start Place:', options=(None, 'Ahiritola', 'Airport', 
'Airport(Gate No. 1)', 'Ajay Nagar…
Dropdown(description='Destination Place:', options=(None, 'Ahiritola', 'Airp
ort', 'Airport(Gate No. 1)', 'Ajay…
Button(description='Calculate', style=ButtonStyle())
Output()
```

2nd mehtod

In [24]:

```
import csv
import heapq
import ipywidgets as widgets
from IPython.display import display, clear_output

# Class representing the graph
class Graph:
    def __init__(self, directed=True):
        self.graph_dict = {}
        self.directed = directed

    def connect(self, A, B, distance=1, difficulty=1):
        adjusted_distance = distance * difficulty  # Adjust distance based c
        self.graph_dict.setdefault(A, {})[B] = adjusted_distance
        if not self.directed:
            self.graph_dict.setdefault(B, {})[A] = adjusted_distance

    def get(self, a):
        return self.graph_dict.get(a, {})

# Function to read data from CSV file
def read_csv(filename):
    with open(filename, 'r', encoding='utf-8') as csvfile:
        return list(csv.DictReader(csvfile))

# Function to extract graph from the read CSV data
def extract_graph(data):
    graph = Graph(directed=False)
    for row in data:
        origin = row['Source']
        destination = row['Destination']
        if destination:
            distance = float(row['Distance(km)']) if row['Distance(km)'] els
            difficulty = float(row.get('Difficulty', 1))  # Assuming a 'Diff
            graph.connect(origin, destination, distance, difficulty)
    return graph

# Dijkstra's algorithm implementation
def dijkstra_search(graph, start, end):
    queue = [(0, start)]
    distances = {start: 0}
    previous_nodes = {start: None}
    visited = set()

    while queue:
        current_distance, current_place = heapq.heappop(queue)
        if current_place in visited:
            continue
        visited.add(current_place)

        if current_place == end:
            path = []
            while current_place is not None:
                path.append(current_place)
                current_place = previous_nodes[current_place]
            return path[::-1], distances[end]

        for neighbor, distance in graph.get(current_place).items():
            new_distance = current_distance + distance
            if neighbor not in visited and (neighbor not in distances or new
                distances[neighbor] = new_distance
                previous_nodes[neighbor] = current_place
                heapq.heappush(queue, (new_distance, neighbor))

    return None, float('inf')

# Main function to create widgets and handle button clicks
def main():
    data = read_csv('data 5.csv')
    graph = extract_graph(data)

    unique_places = sorted(set(row['Source'] for row in data) | set(row['Des
    places_list = [None] + unique_places

    start_place_dropdown = widgets.Dropdown(options=places_list, descriptior
    end_place_dropdown = widgets.Dropdown(options=places_list, description='
    calculate_button = widgets.Button(description='Calculate')
    output = widgets.Output()

    def on_calculate_button_click(b):
        start_place = start_place_dropdown.value
        end_place = end_place_dropdown.value

        with output:
            clear_output()
            if start_place is None or end_place is None:
                print("Error: Please select both start and destination place
            elif start_place == end_place:
                print("Error: Start and destination places cannot be the sam
            else:
                path, total_distance = dijkstra_search(graph, start_place, e
                if path:
                    print(f"Shortest route from {start_place} to {end_place}
                    print(f"Total Path Cost (in Km): {total_distance}")
                else:
                    print("No path found.")

    calculate_button.on_click(on_calculate_button_click)
    display(start_place_dropdown, end_place_dropdown, calculate_button, outp

# Entry point of the program
```

```
if __name__ == "__main__":
    main()
```

```
Dropdown(description='Start Place:', options=(None, 'Ahiritola', 'Airport',
'Airport(Gate No. 1)', 'Ajay Nagar…
Dropdown(description='Destination Place:', options=(None, 'Ahiritola', 'Airp
ort', 'Airport(Gate No. 1)', 'Ajay…
Button(description='Calculate', style=ButtonStyle())
Output()
```

3rd method

In [20]:
```
import csv
import heapq
import ipywidgets as widgets
from IPython.display import display, clear_output

# Class representing the graph
class Graph:
    def __init__(self):
        self.edges = {}

    def connect(self, A, B, distance=1, difficulty=1):
        adjusted_distance = distance * difficulty  # Adjust distance based o
        self.edges.setdefault(A, {})[B] = adjusted_distance
        self.edges.setdefault(B, {})[A] = adjusted_distance  # Assuming undi

    def get_neighbors(self, node):
        return self.edges.get(node, {})

# Function to read data from CSV file
def read_csv(filename):
    with open(filename, 'r', encoding='utf-8') as csvfile:
        return list(csv.DictReader(csvfile))

# Function to extract graph from the read CSV data
def extract_graph(data):
    graph = Graph()
    for row in data:
        origin = row['Source']
        destination = row['Destination']
        if destination:
            distance = float(row['Distance(km)']) if row['Distance(km)'] els
            difficulty = float(row.get('Difficulty', 1))  # Assuming a 'Diff
            graph.connect(origin, destination, distance, difficulty)
    return graph

# BFS with a priority queue to find the shortest path
def bfs_shortest_path(graph, start, end):
    priority_queue = [(0, start)]  # (cost, node)
    visited = set()
    previous_nodes = {start: None}
    distances = {start: 0}

    while priority_queue:
        current_cost, current_node = heapq.heappop(priority_queue)
        if current_node in visited:
            continue
        visited.add(current_node)

        if current_node == end:
            path = []
            while current_node is not None:
                path.append(current_node)
                current_node = previous_nodes[current_node]
            return path[::-1], current_cost

        for neighbor, weight in graph.get_neighbors(current_node).items():
            new_cost = current_cost + weight
            if neighbor not in visited and (neighbor not in distances or new
                distances[neighbor] = new_cost
                previous_nodes[neighbor] = current_node
                heapq.heappush(priority_queue, (new_cost, neighbor))

    return None, float('inf')

# Main function to create widgets and handle button clicks
def main():
    data = read_csv('data 5.csv')
    graph = extract_graph(data)

    unique_places = sorted(set(row['Source'] for row in data) | set(row['Des
    places_list = [None] + unique_places

    start_place_dropdown = widgets.Dropdown(options=places_list, description
    end_place_dropdown = widgets.Dropdown(options=places_list, description='
    calculate_button = widgets.Button(description='Calculate')
    output = widgets.Output()

    def on_calculate_button_click(b):
        start_place = start_place_dropdown.value
        end_place = end_place_dropdown.value

        with output:
            clear_output()
            if start_place is None or end_place is None:
                print("Error: Please select both start and destination place
            elif start_place == end_place:
                print("Error: Start and destination places cannot be the sam
            else:
                path, total_distance = bfs_shortest_path(graph, start_place,
                if path:
                    print(f"Shortest route from {start_place} to {end_place}
                    print(f"Total Path Cost (in Km): {total_distance}")
                else:
                    print("No path found.")

    calculate_button.on_click(on_calculate_button_click)
    display(start_place_dropdown, end_place_dropdown, calculate_button, outp

# Entry point of the program
```

```
if __name__ == "__main__":
    main()
```

```
Dropdown(description='Start Place:', options=(None, 'Ahiritola', 'Airport',
'Airport(Gate No. 1)', 'Ajay Nagar…
Dropdown(description='Destination Place:', options=(None, 'Ahiritola', 'Airp
ort', 'Airport(Gate No. 1)', 'Ajay…
Button(description='Calculate', style=ButtonStyle())
Output()
```

In [ ]:

```

```

In [ ]:

```

```

In [25]:

```
import csv
import heapq
import ipywidgets as widgets
from IPython.display import display, clear_output

# Class representing the graph
class Graph:
    def __init__(self, directed=True):
        self.graph_dict = {}
        self.directed = directed

    def connect(self, A, B, distance=1):
        self.graph_dict.setdefault(A, {})[B] = distance
        if not self.directed:
            self.graph_dict.setdefault(B, {})[A] = distance

    def get(self, a):
        return self.graph_dict.get(a, {})

# Function to read data from CSV file
def read_csv(filename):
    with open(filename, 'r', encoding='utf-8') as csvfile:
        return list(csv.DictReader(csvfile))

# Function to extract graph from the read CSV data
def extract_graph(data):
    graph = Graph(directed=False)
    for row in data:
        origin = row['Source']
        destination = row['Destination']
        if destination:
            distance = float(row['Distance(km)']) if row['Distance(km)'] els
            graph.connect(origin, destination, distance)
    return graph

# Dijkstra's algorithm implementation with a longer distance check
def dijkstra_search(graph, start, end):
    queue = [(0, start)]
    distances = {start: 0}
    previous_nodes = {start: None}
    visited = set()
    while queue:
        current_distance, current_place = heapq.heappop(queue)

        if current_place in visited:
            continue
        visited.add(current_place)

        if current_place == end:
            path = []
            while current_place is not None:
                path.append(current_place)
                current_place = previous_nodes[current_place]
            return path[::-1], distances[end]

        for neighbor, distance in graph.get(current_place).items():
            new_distance = current_distance + distance

            # Prioritize shorter routes, even if they involve more stops
            if neighbor not in visited and (neighbor not in distances or new
                distances[neighbor] = new_distance
                previous_nodes[neighbor] = current_place
                heapq.heappush(queue, (new_distance, neighbor))

    return None, float('inf')

# Main function to create widgets and handle button clicks
def main():
    data = read_csv('data 5.csv')  # Assuming this is your dataset
    graph = extract_graph(data)

    unique_places = sorted(set(row['Source'] for row in data) | set(row['Des
    places_list = [None] + unique_places

    start_place_dropdown = widgets.Dropdown(options=places_list, descriptio
    end_place_dropdown = widgets.Dropdown(options=places_list, description='
    calculate_button = widgets.Button(description='Calculate')
    output = widgets.Output()

    def on_calculate_button_click(b):
        start_place = start_place_dropdown.value
        end_place = end_place_dropdown.value

        with output:
            clear_output()
            if start_place is None or end_place is None:
                print("Error: Please select both start and destination place
            elif start_place == end_place:
                print("Error: Start and destination places cannot be the sam
            else:
                path, total_distance = dijkstra_search(graph, start_place, e
                if path:
                    print(f"Shortest route from {start_place} to {end_place}
                    print(f"Total Path Cost (in Km): {total_distance:.2f}")
                else:
                    print("No path found.")
```

```python
    calculate_button.on_click(on_calculate_button_click)
    display(start_place_dropdown, end_place_dropdown, calculate_button, outp

# Entry point of the program
if __name__ == "__main__":
    main()
```

```
Dropdown(description='Start Place:', options=(None, 'Ahiritola', 'Airport',
'Airport(Gate No. 1)', 'Ajay Nagar…
Dropdown(description='Destination Place:', options=(None, 'Ahiritola', 'Airp
ort', 'Airport(Gate No. 1)', 'Ajay…
Button(description='Calculate', style=ButtonStyle())
Output()
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

This notebook was converted with convert.ploomber.io